**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский Авиационный Институт»**

**(Национальный Исследовательский Университет)**

**Институт: №8 «Информационные технологии   
и прикладная математика»   
Кафедра: 806 «Вычислительная математика   
и программирование»**

Лабораторная работа № 4   
по курсу «Криптография»

Группа: М8О-309Б-22

Студент: О. С. Концебалов

Преподаватель: А. В. Борисов

Оценка:

Дата: 12.04.2025

Москва, 2025

**ОГЛАВЛЕНИЕ**

[1 Тема 3](#_Toc158983147)

[2 Задание 3](#_Toc158983148)

[3 Теория 4](#_Toc158983149)

[4 Ход лабораторной работы 5](#_Toc158983150)

[5 Выводы 6](#_Toc158983151)

# **Тема**

Эллиптические кривые

# **Задание**

Подобрать такую эллиптическую кривую, порядок точки которой полным перебором находится за 10 минут на ПК. Упомянуть в отчёте результаты замеров работы программы, характеристики вычислителя. Также указать какие алгоритмы и/или теоремы существуют для облегчения и ускорения решения задачи полного перебора. Рассмотреть для случая конечного простого поля Z\_p.

# **Теория**

Эллиптическая кривая — это набор точек, описывающихся уравнением Вейерштрасса:



Осталось уточнить всего одну деталь. Все рассмотренные выше кривые относятся к эллиптическим кривым над вещественными числами. И это приводит нас к проблеме округления. Т.е., используя кривые над вещественными числами, мы не сможем получить биекцию между исходным текстом и зашифрованными данными. Чтобы не заморачиваться с округлением в криптографии используются только кривые над конечными полями. Это означает, что под эллиптической кривой понимается набор точек, чьи координаты принадлежат конечному полю.

В криптографии рассматривается два вида эллиптических кривых: над конечным полем — кольцо вычетов по модулю простого числа. И над полем — бинарное конечное поле.

У эллиптических кривых над полем есть одно важное преимущество, элементы поля могут быть легко представлены в виде n-битных кодовых слов, это позволяет увеличить скорость аппаратной реализации эллиптических алгоритмов.

Все математические операции на эллиптических кривых над конечным полем производятся по законам конечного поля, над которым построена эллиптическая кривая. Т.е. для вычисления, например, суммы двух точек кривой E над кольцом вычетов все операции производятся по модулю числа p.

Точки эллиптической кривой над конечным полем представляют собой группу. И как мы отмечали выше для этой группы определена операция сложения.

Соответственно мы можем представить умножение числа k на точку G как G+G+..+G с k слагаемыми.

Теперь представим, что у нас имеется сообщение M представленное в виде целого числа. Мы можем зашифровать его, используя выражение

C=M\*G.

Вопрос в том, насколько сложно восстановить M зная параметры кривой E(a,b), шифротекст С и точку G.

Данная задача называется дискретным логарифмом на эллиптической кривой и не имеет быстрого решения. Более того, считается, что задача дискретного логарифма на эллиптической кривой является более трудной для решения, чем задача дискретного логарифмирования в конечных полях.

# **Ход лабораторной работы**

Первым делом я решил посмотреть характеристики своего домашнего ноутбука, на котором буду производить вычисления. Получил следующие характеристики:

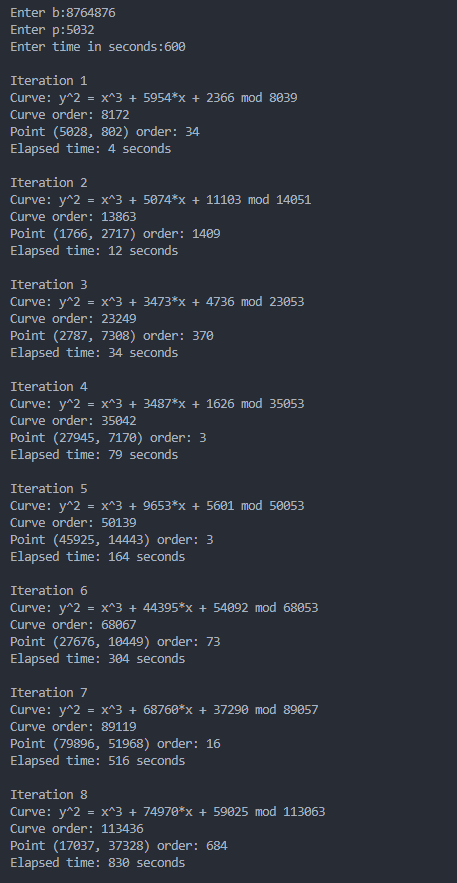
* Процессор - AMD Ryzen 5 5600H with Radeon Graphics, 3301 МГц, ядер: 6, логических процессоров: 12
* Оперативная память – 16 Gb, DDR 4

Во время вычислений ноутбук был подключен к питанию от сети и работал на режиме производительность.

Для выполнения лабораторной я решил использовать язык программирования C++, так как он в разы производительнее других ЯП, таких как Python, Java, Golang и является моим любимым ЯП (хобби). Благодаря этому вычисленная кривая будет еще дольше считаться на других, более распространенных ЯП.

Для выполнения задания лабораторной работы была разработана программа на C++, реализующая базовые операции над эллиптической кривой над конечным полем простого порядка. Пользователь задаёт параметры кривой и время выполнения анализа (время, за которое минимум должна вычисляться кривая). Алгоритм последовательно генерирует все точки кривой, вычисляет её порядок, случайным образом выбирает точку и определяет её порядок. После каждой итерации модуль поля увеличивается до следующего простого числа. Программа замеряет время выполнения каждой итерации для оценки производительности.

Коэффициенты A и B я взял рандомными – просто ввел случайные числа с клавиатуры – A = 2562356; B = 8764876. Коэффициент P = 5032



Выше прикреплен вывод программы – последнее значение и есть наша кривая, которая вычислилась за 830 секунд. Ушло 8 итераций, чтобы вычислить кривую. Так же интересно отметить, что каждая следующая итерация выполнялась примерно в 1,5-2 раза дольше предыдущей.

**Результат:** y2 = x3 + 74970x + 59025

Ниже прикреплен код программы для вычисления кривой:

#include <chrono>

#include <cstdint>

#include <iostream>

#include <random>

#include <tuple>

#include <vector>

struct Point {

    int64\_t x;

    int64\_t y;

    Point(int64\_t *x* = 0, int64\_t *y* = 0) : x(*x*), y(*y*) {}

    bool operator==(const Point& *other*) const {

        return x == *other*.x && y == *other*.y;

    }

    bool operator!=(const Point& *other*) const {

        return !(\*this == *other*);

    }

    friend std::ostream& operator<<(std::ostream& *os*, const Point& *p*) {

*os* << "(" << *p*.x << ", " << *p*.y << ")";

        return *os*;

    }

};

class EllipticCurve {

  private:

    int64\_t A, B, p;

    int64\_t ap, bp;

    std::mt19937\_64 rng;

  public:

    EllipticCurve(int64\_t *A*, int64\_t *B*, int64\_t *P*)

        : A(*A*), B(*B*), p(*P*), rng(2904) {

        ap = *A* % p;

        bp = *B* % p;

    }

    int64\_t getP() const { return p; }

    void setP(int64\_t *P*) {

        p = *P*;

        ap = A % p;

        bp = B % p;

    }

    bool isEllipticCurve(int64\_t *x*, int64\_t *y*) const {

        return (powMod(*y*, 2, p)) == ((powMod(*x*, 3, p) + ap \* *x* + bp) % p);

    }

    int64\_t powMod(int64\_t *base*, int64\_t *exp*, int64\_t *mod*) const {

        int64\_t result = 1;

*base* = *base* % *mod*;

        while (*exp* > 0) {

            if (*exp* % 2 == 1)

                result = (result \* *base*) % *mod*;

*base* = (*base* \* *base*) % *mod*;

*exp* /= 2;

        }

        return result;

    }

    std::vector<int64\_t> extendedEuclideanAlgorithm(int64\_t *a*, int64\_t *b*) const {

        int64\_t s = 0, t = 1, r = *b*;

        int64\_t oldS = 1, oldT = 0, oldR = *a*;

        while (r != 0) {

            int64\_t quotient = oldR / r;

            std::tie(oldR, r) = std::make\_pair(r, oldR - quotient \* r);

            std::tie(oldS, s) = std::make\_pair(s, oldS - quotient \* s);

            std::tie(oldT, t) = std::make\_pair(t, oldT - quotient \* t);

        }

        return { oldR, oldS, oldT };

    }

    int64\_t inverseOf(int64\_t *n*) const {

        auto res = extendedEuclideanAlgorithm(*n*, p);

        int64\_t gcd = res[0], x = res[1];

        if (gcd != 1)

            return -1;

        else

            return (x % p + p) % p;

    }

    Point addPoints(const Point& *p1*, const Point& *p2*) const {

        if (*p1* == Point(0, 0)) return *p2*;

        if (*p2* == Point(0, 0)) return *p1*;

        if (*p1*.x == *p2*.x && *p1*.y != *p2*.y) return Point(0, 0);

        int64\_t s;

        if (*p1* == *p2*) {

            int64\_t numerator = (3 \* powMod(*p1*.x, 2, p) + ap) % p;

            int64\_t denominator = inverseOf(2 \* *p1*.y);

            if (denominator == -1) return Point(0, 0);

            s = (numerator \* denominator) % p;

        } else {

            int64\_t numerator = (*p1*.y - *p2*.y + p) % p;

            int64\_t denominator = inverseOf((*p1*.x - *p2*.x + p) % p);

            if (denominator == -1) return Point(0, 0);

            s = (numerator \* denominator) % p;

        }

        int64\_t x = (powMod(s, 2, p) - 2 \* *p1*.x + p) % p;

        int64\_t y = (*p1*.y + s \* (x - *p1*.x + p)) % p;

        return Point(x, (p - y) % p);

    }

    int64\_t orderPoint(const Point& *point*) const {

        int64\_t i = 1;

        Point check = addPoints(*point*, *point*);

        while (check != Point(0, 0)) {

            check = addPoints(check, *point*);

            ++i;

        }

        return i + 1;

    }

    int64\_t step() {

        std::cout << "Curve: y^2 = x^3 + " << ap << "\*x + " << bp << " mod " << p << "\n";

        std::vector<Point> points;

        auto start = std::chrono::high\_resolution\_clock::now();

        for (int64\_t x = 0; x < p; ++x) {

            for (int64\_t y = 0; y < p; ++y) {

                if (isEllipticCurve(x, y))

                    points.emplace\_back(x, y);

            }

        }

        std::cout << "Curve order: " << points.size() << "\n";

        std::uniform\_int\_distribution<int64\_t> dist(0, points.size() - 1);

        Point randomPoint = points[dist(rng)];

        std::cout << "Point " << randomPoint << " order: " << orderPoint(randomPoint) << "\n";

        auto end = std::chrono::high\_resolution\_clock::now();

        int64\_t elapsedTime = std::chrono::duration\_cast<std::chrono::seconds>(end - start).count();

        std::cout << "Elapsed time: " << elapsedTime << " seconds\n\n";

        return elapsedTime;

    }

    bool isPrimeNumber(int64\_t *number*) const {

        if (*number* < 2) return false;

        for (int64\_t i = 2; i \* i <= *number*; ++i) {

            if (*number* % i == 0)

                return false;

        }

        return true;

    }

    int64\_t getNextPrimeNumber(int64\_t *start*) const {

        while (!isPrimeNumber(*start*)) {

            ++*start*;

        }

        return *start*;

    }

};

int main() {

    int64\_t a;

    int64\_t b;

    int64\_t p;

    int64\_t timeToCalculate;

    std::cout << "Enter a:";

    std::cin >> a;

    std::cout << "Enter b:";

    std::cin >> b;

    std::cout << "Enter p:";

    std::cin >> p;

    std::cout << "Enter time in seconds:";

    std::cin >> timeToCalculate;

    std::cout << std::endl;

    EllipticCurve ec(a, b, p);

    int64\_t timePassed = 0;

    int64\_t iter = 1;

    while (timePassed < timeToCalculate) {

        std::cout << "Iteration " << iter << "\n";

        ec.setP(ec.getNextPrimeNumber(ec.getP() + iter \* 3000));

        timePassed = ec.step();

        ++iter;

    }

    return 0;

}

# **Выводы**

В ходе выполнения лабораторной работы я приобрел знания об эллиптических кривых, а также с помощью полного перебора вычислил такую кривую за ~830 секунд (еще учитываем погрешность). Было здорово наконец-то вернуться на плюсы, с которыми уже давно не доводилось поработать.

Для ускорения решения задачи полного перебора порядка точки на эллиптической кривой над конечным полем можно использовать: теорему Лагранжа, согласно которой, порядок любой точки P на эллиптической кривой делит порядок группы точек кривой; алгоритм Шенкса, который позволяет эффективно находить порядок циклической группы за время, пропорциональное квадратному корню из порядка группы; алгоритм Полларда, который может находить порядок точки в произвольной конечной абелевой группе; использование параллельных вычислений; алгоритм Шуфа.

# **Список используемой литературы**

* <https://habr.com/ru/articles/335906/>
* <https://royalforkblog.github.io/2014/09/04/ecc/>
* <https://znanierussia.ru/articles/Эллиптическая_кривая>
* <https://eax.me/elliptic-curves-crypto/>